Introduction to the <TextCoop> platform and Dislog language, User Manual V.1

Patrick Saint-Dizier,
IRIT - CNRS,
118 route de Narbonne, 31062 Toulouse Cedex, France.
stdizier@irit.fr

April 2012

Table des matières

1 Foundational Aspects of <TextCoop> and Dislog **4**
- 1.1 The Challenges . . . 4
- 1.2 The <TextCoop> platform and the Dislog language . . . 5
  - 1.2.1 Some linguistic considerations . . . 5
  - 1.2.2 Some foundational principles of <TextCoop> . . . 5
  - 1.2.3 The structure of Dislog rules . . . 6
  - 1.2.4 Dislog advanced features . . . 8
  - 1.2.5 Introducing reasoning aspects into discourse analysis . . . 10
  - 1.2.6 Processing complex constructions : the case of 'Dislocation' constructions . . . 11
- 1.3 The <TextCoop> engine . . . 11
  - 1.3.1 System performances and discussion . . . 12
  - 1.3.2 The <TextCoop> environment . . . 14

2 Writing Dislog rules **15**
- 2.1 Instruction . . . 15
- 2.2 Advice . . . 16
- 2.3 Warning . . . 16
- 2.4 Binding rules for warnings . . . 17
- 2.5 Cause . . . 17
- 2.6 Condition . . . 17
- 2.7 Concession . . . 18
- 2.8 Contrast . . . 18
- 2.9 Circumstance . . . 19
- 2.10 Purpose . . . 19
- 2.11 Illustration . . . 19
- 2.12 Illustration : a simple example of a reasoning schema for binding purposes . . . 20
- 2.13 Restatement . . . 20
- 2.14 The Art of writing Dislog rules and constraints . . . 21

3 Using Dislog and TextCoop **23**
- 3.1 A few warnings before starting . . . 23
- 3.2 Installation . . . 23
- 3.3 Starting . . . 24
- 3.4 Writing rules . . . 24

3.4.1 Writing a rule in Dislog 25
3.4.2 Writing context-dependent rules 27
3.4.3 Parameters and Structure Declarations 27
3.4.4 Lexical data 28
3.4.5 Other resources 29
3.4.6 Input-Output management 30
3.5 Execution schema and structure of control 30

Warning

This document is a preliminary version of a user manual for the <TextCoop> platform and the Dislog language. The code which is distributed is also a kind of **pre-beta** version. It has however been tested without any problem on a number of types of texts with a large diversity of rules.

The software which is delivered at the moment is a kernel. It does contain the reasoning facilities module, but at that moment no predefined function is given (but the user may include his own), a well-developed environment and a rule management system (that would e.g. control the rule syntax, optimize them or translate them into the Dislog format). These modules will come with a second release of the code which will be made available later.

However the system can be used as it is in this version. It has in fact been used by several of our students in linguistics, with little knowledge in computer science and in Prolog in particular.

The <TextCoop> platform and Dislog language run on different systems, in particular Windows and Linux. Prolog, e.g. SWI Prolog, must be installed first. A basic knowledge of Prolog is preferable (syntax and execution schema).

We suggest the reader to first read Chapters 1 and 2 of this document which give some foundations and background. Chapter 3 essentially describes how to use the main features of Dislog and TextCoop.

We are aware that this first version has many imperfections and limitations. However, its use in several large size industrial applications shows that it is viable. We thank our users in advance for any comments, questions and suggestions they could send us.

Linguistic resources which are provided here are rather simple and given for the purpose of illustration. We have however developed much larger sets of rules, in particular around argumentation and explanation structures. These can be made available upon request. However a good knowledge of the tool is necessary for their full understanding.

Chapitre 1

Foundational Aspects of <TextCoop> and Dislog

This chapter summarizes the concepts at the basis of Dislog and how they are interpreted in the <TextCoop> engine. The strategy adopted in Dislog is that basic discourse units and functions (e.g. illustration) are recognized by means of rules. Then, with the same formalism, another device, bounding rules, is introduced to bind basic units into larger structures. There is no theoretical approach taken a priori on discourse analysis in Dislog. It is an hopefully convenient logic-based programming language to recognize discourse structures in texts. Finally Dislog includes a variety of devices to express constraints and to introduce knowledge and reasoning. These elements are introduced hereafter.

Chapter 2 develops the linguistic analysis of a number of discourse functions, while Chapter 3 shows how these are implemented in Dislog.

1.1 The Challenges

Discourse analysis[1] is a very challenging task because of the large diversity of discourse structures, the various forms they take in language and the potential knowledge needs for their identification. Rhetorical structure theory (RST) (Mann el al. 1988) is a major attempt to organize investigations in discourse analysis, with the definition of 22 basic structures. Since then, almost 200 relations have been introduced with various aims `http ://www.sfu.ca/rst/`. Several approaches, based on corpus analysis with a strong linguistic basis, are of much interest for our aims. Relations are investigated together with their linguistic markers e.g. (Delin 1994), (Marcu 1997), (Miltasaki et ali. 2004), then (Kosseim et al. 2000) for language generation, and (Rossner et al. 1992), and (Saito et al. 2006) with an extensive study on how markers can be quite systematically acquired.

TextCoop is a logic-based platform designed to describe and implement discourse structures and related constraints via an authoring tool. Dislog (Discourse in Logic) is the language designed for writing rules and lexical data. Dislog extends the formalism of Definite Clause Grammars and Metamorphosis Grammars (since type 1 expressions are allowed) to discourse

[1] This first chapter is a revision of our LREC 2012 paper on TextCoop foundations

processing and allows the integration of knowledge and inferences. TextCoop and Dislog tackle the following foundational and engineering problems :

- taking into account of the diversity of discourse structures : generic (e.g. illustration, elaboration) as well as domain oriented (e.g. title-instructions in procedures),
- introduction, for easy tests and updates, of a declarative and modular language via rules. Our approach is based on (1) basic discourse structures, (2) selective binding rules to bind basic structures into larger units, (3) repair rules, (4) various classes of constraints on the way basic structures can be combined and (5) reasoning procedures.
- introduction of accurate specifications of rule execution modes (e.g. order, concurrency, left-to-right or right-to-left, etc.), in order to optimally process structures,
- taking into account of the specification and binding of complex structures, e.g. multi-nucleus-satellite constructions as often found in domain dependent constructions (e.g. title-prerequisites-instructions in procedures), or cases where satellites are merged into their nucleus (dislocation),
- integration in rules of various forms of knowledge and inferences e.g. to compute attribute values or to resolve relation identification and scope, or ambiguities between various relations.
- development of an authoring tool to implement discourse relation rules and lexical resources. Note that in general discourse analysis rules are relatively re-usable over domains because markers are often domain-independent.
- finally, production of various forms of output representations (XML tags, dependencies).

1.2 The <TextCoop> platform and the Dislog language

1.2.1 Some linguistic considerations

Most works dedicated to discourse analysis have to deal with the triad : discourse function identification, delimitation of its textual structure (boundaries of the discourse unit) and structure binding. By function we mean a nucleus or a satellite of a rhetorical relation or similar structure, e.g. an illustration, an illustrated expression, an elaboration, or the elaborated expression, a conditional expression, a goal expression, etc. Functions are realized by textual structures which need to be accurately delimited. Functions are not stand alone : they must be bound based on the nucleus-satellite or nucleus-nucleus principle.

1.2.2 Some foundational principles of <TextCoop>

The necessity of a modular approach, open to the diversity of language constructs, where each aspect of discourse analysis is dealt with accurately and independently in a module, while keeping open all the possibilities of interaction, if not concurrency, between modules has lead us to consider some simple elements of the model of Generative Syntax (a good synthesis is given in (Lasnik et al. 1988)). As shall be seen below, we introduce :

- productive principles, which have a high level of abstraction, which are linguistically sound, but which may be too powerful,
- restrictive principles, which limit the power of the first in particular on the basis of well-formedness constraints.

Another foundational feature is an integrated view of markers used to identify discourse functions, merging lexical objects with morphological functions, typography and punctuation, syntactic constructs, semantic features and inferential patterns that capture various forms of knowledge (domain, lexical, textual). <TextCoop> is the first platform that offers this view within a logic-based approach. If machine learning is a possible approach for sentence processing, where interesting results have emerged, it seems not to be so successful for discourse analysis (e.g. Carlson et ali. 2001), (Saaba et al 2008)). This is due to two main factors : (1) the difficulty to annotate discourse functions in texts and the high level of disagreement between annotators and (2) the large non-determinism of discourse structure recognition where markers are often immerged in long spans of text of no or little interest. For these reasons, we adopted a rule-based approach. Rules are hand coded, based on corpus analysis using bootstrapping tools. They may also emerge from automatic learning procedures.

Dislog rules basically implement the productive principles. They are composed of three main parts :

1. A discourse function (or unit) identification structure, which basically has the form of a rule or of a pattern,
2. A set of calls to inferential forms using various types of knowledge, these forms are part of the identification structure, they may contribute to solving ambiguities, they may also be involved in the computation of the resulting representation or they may lead to restrictions.
3. A structure that represents the result of the analysis : it can be a simple XML structure, or any other structure a priori such as an element of a graph or a dependency structure. More complex representations, e.g. based on primitives, can be computed using a rich semantic lexicon.

1.2.3 The structure of Dislog rules

Let us now introduce in more depth the structure of Dislog rules. Dislog follows the principles of logic-based grammars as implemented three decades ago in a series of formalisms, among which, most notably : Definite Clause Grammars (Pereira and Warren 1981), Metamorphosis Grammars (Colmerauer 1978) and Extraposition Grammars (Pereira 1981). These formalisms were all designed for sentence parsing with an implementation in Prolog via a meta-interpreter or a direct translation into Prolog (Saint-Dizier 1994). The last two formalisms include a device to deal with long distance dependencies.

Dislog adapts and extends these grammar formalisms to discourse processing, it also extends the regular expression format which is often used as a basis in language processing tools. The rule system of Dislog is viewed as a set of productive principles.

A rule in Dislog has the following general form, which is globally quite close to Definite Clause Grammars in its spirit :
L(Representation) $\rightarrow$ R, $\{P\}$.
where :

1. L is a non-terminal symbol.
2. Representation is the representation resulting from the analysis, it is in general the original text with XML tags that annotates the discourse structures. It can also be a partial

dependency structure or a more formal representation. This field is indeed totally open and up to the rule author. The computation of the representation is typical of logic-based grammars and use the power of logic variables of logic programming.

3. R is a sequence of symbols as described below, and
4. P is a set of predicates and functions implemented in Prolog that realize the various computations and controls, and that allow the inclusion of inference and knowledge into rules.

R is a finite sequence of the following elements :

– terminal symbols that represent words, expressions, punctuations, various existing html or XML tags. They are included between square brackets,
– preterminal symbols : are symbols which are derived directly into terminal elements. These are used to capture various forms of generalizations, facilitating rule authoring and update. Symbols can be associated with a type feature structure that encodes a variety of aspects of those symbols, from morphology to semantics,
– non-terminal symbols, which can also be associated with type feature structures. These symbols refer to 'local grammars', i.e. grammars that encode specific syntactic constructions such as temporal expressions or domain specific constructs. Non-terminal symbols do not include discourse structure symbols : Dislog rules cannot call each other, this feature being dealt with by the selective binding principle, which includes additional controls. A rule in Dislog thus basically encodes the recognition of a discourse function taken in isolation, i.e. an elementary discourse unit.
– optionality and iterativity markers over non-terminal and preterminal symbols, as in regular expressions,
– gaps, which are symbols that stand for a finite sequence of words of no present interest for the rule which must be skipped. A gap can appear only between terminal, preterminal or non-terminal symbols. Dislog offers the possibility to specify in a gap a list of elements which must not be skipped : when such an element is found before the termination of the gap, then the gap fails.
– a few meta-predicates to facilitate rule authoring.

Symbols may have any number of arguments. However, in our current version, to facilitate the implementation of the meta-interpreter and improve its efficiency, the recommended form is :
identifier(Representation, Type feature structure).
where Representation is the symbol's representation. In Prolog format, a difference list (E,S) is added at the end of the symbol :

```
identifier(R, TFS, E,S)
```

A few examples in Dislog format are given in Chapter 3. Rules in external format, illustrating the above definitions, can be found in Chapter 2.

Similarly to DCGs and to Prolog clause systems, it is possible and often necessary to have several rules to describe the different realizations of a given discourse function. These all have the same identifier *L*, as it would be the case e.g. for NPs or PPs. A set of rules with the same identifier is called a **cluster of rules**. Rule clusters are executed sequentially by the <TextCoop> engine following an order given in a **cascade**.

1.2.4 Dislog advanced features

Selective binding rules

Selective binding rules allow to link two or more already identified discourse basic units. The objective is e.g. to bind a nucleus with a satellite (e.g. an argument conclusion with its support) or with another nucleus (e.g. a concessive or parallel structures). Selective binding rules can be used for other purposes than implementing rhetorical relations. These can be used more generally to bind structures whose rhetorical status is not so straightforward, in particular in some application domains. For example, in procedural discourse, they can be used to link a title with the set of instructions, prerequisites and warnings that realize the goal expressed by this title.

From a syntactic point of view, selective binding rules are expressed using the Dislog language formalism. Selective binding rules is the means offered by Dislog to construct hierarchical discourse structures from the elementary ones identified by the rule system. Different situations occur that make binding rules more complex than any system of rules used for sentence processing, in particular (examples are given in section 2.6) :

- discourse structures may be embedded to a high degree, with partial overlaps,
- others may be chained (a satellite is a nucleus for another relation),
- nucleus and related satellites may be non-adjacent,
- nucleus may be linked to several satellites of different types,
- some satellites may be embedded into their nucleus.

Selective binding rules allow the binding of :

1. two adjacent structures, in general a nucleus and a satellite, or another nucleus.
2. two or more non-adjacent structures, which may be separated by various elements (e.g. causes and consequences, conclusion and supports of arguments may be separated by various considerations). However limits must be imposed on the 'textual distance' between units.

To limit the textual distance between argument units, we introduce the notion of **bounding node**, which is also a notion used in sentence formal syntax to restrict the way long-distance dependencies can be established (Lasnik et al. 1988). Bounding nodes are also defined in terms of barriers in generative syntax. In our case, the constraint is that a gap must not go over a bounding node. This allows to restrict the distance between the constituents which are bound. For example, we consider that an argument conclusion and support must be both in the same paragraph, therefore, the node 'paragraph' is a bounding node.

This declaration is taken into account by the <TextCoop> engine in a transparent way, and interpreted as an active constraint which must be valid throughout the whole parsing process. The situation is however more complex than in sentence syntax. Indeed, bounding nodes in discourse depend on the structure being processed. For example, in the case of procedural discourse, a warning can be bound to one or more instructions which are in the same subgoal structure. Therefore, the bounding node must be the subgoal node, which may be much larger than a paragraph. Bounding nodes are declared as follows in Dislog :

```
boundingNode(paragraph, argument).
```

Repair rules

Although relatively unusual, when parsing or computing representations, annotation errors may occur. This is in particular the case when (1) a rule has a fuzzy or ambiguous ending condition w.r.t. the text being processed or (2) when rules of different discourse functions overlap, leading to closing tags that may not be correctly inserted. In argument recognition, we have indeed some forms of competition between a conclusion and its support which share common linguistic markers. For example, when there are several causal connectors in a sentence the beginning of a support is ambiguous since most supports are introduced by a connector. In addition to using concurrent processing strategies, repair rules can resolve errors efficiently.

The most frequent situation is the following :
<a>, ...<b> < /a>, ... < /b>
which must be rewritten into :
<a>, ... < /a>, ... <b>, ... < /b> .
This is realized by the following rule :

```
correction([<A> G1 </A> G2 <B> G3 </B>]) -->
     [<A>], gap(G1),[<B>],gap(G2),
        [</A>],  gap(G3), [</B>].
```

Rule concurrency management

The current <TextCoop> engine is close to the Prolog execution schema. However, to properly manage rule execution but also the properties of discourse structures and the way they are usually organized, we introduce additional constraints, which are, for most of them, borrowed from sentence syntax.

Within a cluster of rules, the execution order is the rule reading order, from the first to the last one. Then, elementary discourse functions are first identified and then bound to others to form larger units, via selective binding rules. Following the principle that a text unit has one and only one discourse function (but may be bound to several other structures via several rhetorical relations) and because rules can be ambiguous from one cluster to another, the order in which rule clusters are executed is a crucial parameter. There is indeed no backtracking over previous elements in the cluster to revise a parse that has succeeded. In our engine, there is no backtracking between clusters. To handle this strategy, Dislog requires that rule clusters are executed in a precise, predefined order, implemented in a **cascade of clusters of rules**.

For example if, in a procedure, we want first titles, then prerequisites and then instructions to be identified, the following constraint must be specified :

```
title < prerequisite < instruction.
```

Since titles have almost the same structure than instructions, but with additional features (bold font, html specific tags, etc.), this prevents titles from being erroneously identified as instructions.

In relation with this notion of cascade, it is possible to declare **'closed zones'**, e.g. :

```
closed_zone([title]).
```

indicates that the textual span recognized as a title must not be considered again to recognize other functions within or over it (via a gap).

Structural constraints

Let us now consider basic structural principles, which are very common in language syntax. This allows us to contrast the notion of consistuency with the notion of discourse relation. Consistuency is basically a part-of relation applied to language structures (nouns are part of NPs) while discourse is basically relational. Let us introduce here dominance and precedence constraints, these notions being valid as far as trees of discourse structures can be constructed, which is in fact the most frequent situation. Discourse abound in various types of constraints, which may be domain dependent. Dislog is open to the specification of a number of such structural constraints. These are interpreted by the meta-interpreto in <TextCoop> as active constraints.

Dominance constraints can be stated as follows :

```
dom(instruction, condition).
```

This constraint, where instruction and condition are two cluster names, states that a conditional expression is always dominated by an instruction, i.e. the condition XML tags are strictly within the boundaries of an instruction XML tags. This means that a condition must always be part of an instruction, not in a discourse relation with an instruction. In that case, there is no discourse link between a condition and an instruction, the implicit structure being consistuency : a condition is a constituent, or a part of, an instruction.

Similarly, **non-dominance constraints** can be stated to ensure that two discourse functions appear in different branches of the discourse representation, e.g. :

```
not_dom(instruction, warning).
```

which states that an instruction cannot dominate a warning.

Finally, **precedence constraints** may be stated. We only consider here the case of immediate linear precedence, for example :

```
prec(elaborated, elaboration).
```

indicates that an elaboration must follow what is elaborated. This is a useful constraint for the cases where a nucleus must necessarily precede its satellite : it contributes to the efficiency of the selective binding mechanism and resolves some recognition ambiguities.

1.2.5 Introducing reasoning aspects into discourse analysis

Discourse relation identification often require some forms of knowledge and reasoning. This is in particular the case to resolve ambiguities in relation identification when there are several candidates or to clearly identify the text span at stake. While some situations are extremely difficult to resolve, others can be processed e.g. via lexical inference or reasoning over ontological knowledge. Dislog allows the introduction of reasoning, and the <TextCoop> platform allows the integration of knowledge and functions to access it and reason about it.

This problem is very vast and largely open, with exploratory studies e.g. reported in (Van Dijk 1980), (Kintsch 1988), and more recently some debates reported in :
(http ://www.discourses.org/Unpublished Articles/SpecDis&Know.htm) .

Let us give here a simple motivational example. The utterance (found in our corpus) :
... red fruit tart (strawberries, raspberries) are made ...
contains a structure : (*strawberries, raspberries*) which is ambiguous in terms of discourse functions : it can be an elaboration or an illustration, furthermore the identification of its nucleus is ambiguous :
red fruit tart, red fruit ?
A straightforward access to an ontology of fruits tells us that those berries are red fruits, therefore :
- the unit *strawberries, raspberries* is interpreted as an illustration, since no new information is given (otherwise it would have been an elaboration)
- its nucleus is the '*red fruit*' unit only,
- and it should be noted that these two constituents, which must be bound, are not adjacent.

The relation between an argument conclusion and its support may not necessarily straightforward and may involve various types of domain and common-sense knowledge :
do not park your car at night near this bar : it may cost you fortunes.
Women standards of living have progressed in Nepal : we now see long lines of young girls early morning with their school bags. (Nepali Times).
In this latter example, *school bag* means going to school, then *school* means education, which in turn means better conditions, for women in this case.

Predefined reasoning functiosn are not yet available, however the rule writer can define his own.

1.2.6 Processing complex constructions : the case of 'Dislocation' constructions

As in any language situation, there are complex situations where discourse segments that contribute to form larger units, which are not clearly delimited, may overlap, be shared by several discourse relations, etc. Similarly to syntax, we identified in relatively 'free style' texts phenomena similar to quasi-scrambling situations, free-structure ordering and cleft constructions.

From a processing point of view, the <TextCoop> engine attempts to recognize the embedded structure first, then, if no unique text segment can be found for the embedding structure (standard case), it non-deterministically decomposes the rules describing the embedding structure one after the other, following the above constraints, and attempts to recognize it 'around' the embedded one.

As an example, we observed in our corpora quasi-scrambling situations, a simple case being the illustration relation. Consider again the example above, which can also be written as follows (in French) :
strawberries are red fruits similarly to raspberries, for example.
where the enumeration itself is subject to dislocation.

1.3 The <TextCoop> engine

Let us now give some details about the way the <TextCoop> engine runs. The engine and its environment are implemented in SWI Prolog, using the standard Prolog syntax without using any libraries to guarantee readability, ease of update and portability. Since this is quite a

complex implementation, we simply survey here the elements which are crucial for our current purpose. The principle is that the declarative character of constraints and structure processing and building is preserved in the system. The engine, implemented in Prolog, interprets them at the appropriate control points.

The constraints advocated above remain as given in the examples below, these are directly consulted by the meta-interpreter to realize the relevant controls. The engine follows the cascade specification for the execution of rule clusters. For each cluster, rules are activated in their reading order, one after the other. Backtracking manages rule failures. If a rule in a rule cluster succeeds on a given text span, then the other possibilities for that cluster are not considered (but rules of other clusters may be considered in a later stage of the cascade).

A priori, the text is processed via a left to right strategy. In a cluster of rules, rules are executed sequentially, however, if a rule starts with an early symbol (e.g. a determiner), it is activated before another rule that starts on a later symbol (e.g. the noun it quantifies). <TextCoop> also offers a right to left strategy for rules where the most relevant markers are to the right of the rule, in order to limit backtracking. For the two types of readings, the system is tuned to recognize the smallest text span that satisfies the rule structure.

It processes raw text, html or XML texts. A priori, the initial XML structure is preserved.

1.3.1 System performances and discussion

Let us now analyze the performances of <TextCoop> with respect to relevant linguistic dimensions.

General results

The <TextCoop> engine and related data are implemented in SWI Prolog which runs on a number on environments (Windows, Linux, Apple). Our implementation can support a multi-threaded approach, which has been tested with the <TextCoop> engine embedded into a Java environment. This is also useful for example for 'parallel' processing on several machines or to distribute e.g. lexical data, grammars and domain knowledge on various machines.

The <TextCoop> engine has been relatively optimized and some recommendations for writing rules have been produced in order to allow for a reasonable efficiency.

Lexical issues

An important feature of discourse structure recognition is that the lexical resources which are needed are quite often generic. This means that the system can be deployed on any application domain without any major lexical changes and update (see chapter 2). In total, the average size of the required lexical resources (number of rules being fixed) for discourse processing for an application such as procedural text parsing on a given domain is around 900 words, which is very small compared to what is necessary to process the structure of sentences for the same domain. Results below are given fro French. Results for English are not very different.

The following figures give the system performances depending on the lexicon size. Lexicon sizes correspond to comprehensive lexicons for a given domain (e.g. 400 corresponds to the cooking domain, the case with 180 lexical entries is a toy system).

lexicon size (in no. of words)	Mb of text/hour
180	39
400	27
900	20
1400	18
2000	17

Fig. 1 Impact of lexicon size

These results are somewhat difficult to precisely analyze, since they depend on the number of words by syntactic category, the way they are coded and the order in which they are listed in the lexicon (in relation with the Prolog strategy). In order to limit the complexity related to morphological analysis, a crucial aspect for Romance languages, a preliminary tagging process has been carried out to limit backtracking. The way lexical resources are used in rules is also a parameter which is difficult to precisely analyse.

Globally, reducing the size of the lexicon to those elements which are really needed for the application allows for a certain increase in the system performances.

Issues related to the rule system size and complexity

Two parameters related to the rule system are investigated here : how much the number of rules and the rule size impact the efficiency.

The results obtained concerning the number of rules are the following :

number of rules	Mb of text/hour
20	29
40	23
70	19
90	18

Fig. 2 Impact of number of rules

As can be noted, increasing the number of rules has a moderate impact on performances, one of the reasons is that the most prototypical rules are executed first. Rules have here an average complexity : 4 symbols and a gap in average, and an average of 8 rules per cluster. Lexical size here is fixed (500 entries). 20 rules is a very small system while 80 to 120 rules is a standard size for an application. The results we obtain are difficult to accurately analyze : besides rule ordering considerations, results depend on the distribution of rules per cluster and the form of the rules. For example, the presence of non-ambiguous linguistic markers at the beginning of a rule enhances rule selection, and therefore improves efficiency. Constraints such as those presented above are also very costly since they are checked at each step of the parsing process for the structures at stake. Selective binding rules have little impact on efficiency : their first symbol being an XML tag backtracking occurs at an early stage of the rule traversal.

Let us now consider rule size, which is obviously an important feature :

rule complexity (symbols per rule)	Mb of text/h
3	30
4	23
5	20
7	18

Fig. 3 Impact of rule size

With the number of rules and the size of the lexicon being kept fixed, we note that the rule size has a moderate impact on performances, slightly higher than the number of rules. This may be explained by the fact that the symbols starting the rules are in a number of cases sufficiently well differentiated to provoke early backtracking if the rule is not the one that must be selected. However, the number of lexical entries of these symbols may have an important impact. If the symbol is a specific type of connector or if it is a noun or a verb, this may entail efficiency differences, difficult however to evaluate at our level. Finally, note that rules have in general between 4 and 6 symbols including gaps.

1.3.2 The <TextCoop> environment

The <TextCoop> environment is in a very early stage of development : many more experiments are needed before reaching a stable analysis of the needs. It includes tools for rules (syntax checking, but also e.g. controlling possible overlaps between rules, bootstrapping on corpora to induce rules) and for developing the necessary lexical resources. Accessing already defined and formatted resources is of much interest for authors. We have already designed the following sets of resources, for French and English. These are not fully included in this first version, but will be given in a second release. Resources are the following :

- lists of connectors, organized by general types : temporal, causal, concession, etc.
- list of specific terms which can appear in a number of discourse functions, e.g. : terms specific of illustration, summarization, reformulation, etc.
- lists of verbs organized by semantic classes, close to those found in WordNet, that we have adapted or refined for discourse analysis, e.g. with a focus e.g. on propositional attitude verbs, report verbs, (Wierzbicka 1987), etc.
- list of terms with positive or negative polarity, essentially adjectives, but also some nouns and verbs, this is useful in particular to evaluate the strength of arguments,
- local grammars for e.g. : temporal expressions, expression of quantity, etc.
- some already defined modules of discourse function rules to recognize general purpose discourse functions such as illustration, definition, reformulation, goal and condition.
- some predefined functions and predicates to access knowledge and control features (e.g. subsumption),
- morphosyntactic tagging functions,
- some basic utilities for integrating knowledge (e.g. ontologies) into the environment.

Chapitre 2

Writing Dislog rules

In this chapter, we present some examples of Dislog rules which can recognize basic discourse structures, such as instruction, illustration, reformulation, etc. We also give examples of binding rules[1]. Examples remain simple, possibly adhoc : they are given as illustrations. The rule sample given here is not in Dislog, it is in a readable form, convenient for linguistic analysis. These rules must then be translated into Dislog (Chapter 3). For the time being there is no automatic translator, as it is the case e.g. for DCGs, therefore a manual encoding must be carried out by the programmer. This is however a rather simple task (and will be available in the next release). In the next chapter we show how these rules are implemented in Dislog.

For each discourse relation, a definition is given in addition to the examples of rules and resources. Rules are designed for French or English, depending on cases. In general, structures are relatively similar. Then some linguistic realizations of discourse relations coming from our corpus of English didactic texts or procedures are provided. The curly brackets show that an element is optional. Resources given here are samples.

The rules presented below have been produced manually, from a manual analysis of discourse structures over a development corpus. We feel this is the best way to proceed for discourse structures. However, rules can result from various types of statistical analysis, including a variety of learning methods. Dislog is general enough to allow various forms of encodings.

Similarly, in the next chapter, we show how to produce representations based on XML tags. This is probably the simplest representation which can be produced. However, Dislog is flexible enough to allow for the production of other types of representations, such as dependencies.

2.1 Instruction

Definition : An instruction in a procedure is a statement, often in an imperative form, that asks to realize an action. This action can possibly be associated with various elements such as instruments, equipments, manners, etc. The main verb of an instruction is often in the imperative or infinitive form in French, in the infinitive form in English.

A few structures :

`Advice` →

```
gap(not(neg), verb(action, infinitive), gap, eos. /
```

[1]This chapter is a part of a paper presented at LREC 2012, (Bourse and Saint-Dizier)

`gap(not(neg), verb(_, faire), gap, verb(action, infinitive), gap, eos.`

'infinitive' denotes a veb in the infinitive form (without 'to'), 'faire' is a light verb in French, 'action' denotes an action verb, which is in general domain dependent. 'eos' denotes end of sentence (via a punctuation mark or any other mark).

Resources :

Besides modals and a few terms like pronouns, the main resource is a list of action verbs. However, in most cases, only a limited set of verbs is needed, about 100.

Example :

Write titles in bold font.

2.2 Advice

Definition : Relation between a conclusion and a support, the conclusion invites the reader to perform an optional action to obtain better results, while the support gives a motivation for realizing this action.

Structures :

`Advice` →
`verb(pref,infinitive), gap(G), eos. /`
`[it,is], adv_prob, gap(G1), exp(advice1), gap(G2), eos. /`
`exp(advice2), gap(G), eos.`

Resources :

verb(pref) : *choose, prefer*
exp(advice1) : *a good idea, better, recommended, preferable*
exp(advice2) : *a X tip, a X advice, best option, alternative*
adv_prob : probably, possibly, etc.

Examples :

Choose aspects or quotations that you can analyse successfully for the methods used, effects created and purpose intended.
Following your thesis statement, it is a good idea to add a little more detail that acts to preview each of the major points that you will cover in the body of the essay.
A useful tip is to open each paragraph with a topic sentence.

2.3 Warning

Definition : Relation between a conclusion and a support, the conclusion drawing the attention of the reader to an action which is compulsory to perform, and the support giving a motivation for realizing this action or the risks which may arise.

Structures :

`Warning-conclusion` →
`exp(ensure), gap(G), eos. /`
`[it,is], adv(int), adj(imp), gap(G), verb(action,infinitive),`
`gap(G), eos.`

Resources :

exp(ensure) : *ensure, make sure, be sure*
adv(int) : *very, absolutely, really*
adj(imp) : *essential, vital, crucial, fundamental*

Examples :

Make sure your facts are relevant rather than related.
It is essential that you follow the guidelines for each proposal as set by the instructor.

2.4 Binding rules for warnings

Let us give here a simple example of a binding rule. Warnings are composed of a conclusion and a support (not developed above). These two structures are recognized separetely by dedicated rules. Then, it is necessary to bind these two structures to get a warning. Let us assume that both supports and conslusions are explicitly tagged, then, a simple binding rule is :

`Warning` $\rightarrow$
`<warning-concl>,gap(G1),</warning-concl>, gap(G2),`
`<warning-supp>, gap(G3), </warning-supp>, gap(G4), eos.`

Then, the whole structure is tagged e.g. <warning>. Similar rules are defined to bind nucleus with their related satellites. G1, G2, G3 and G4 are variables that represent the contents (list of words) skipped by each gap. For different gaps, variables must be different.

Binding rules can be more complex than this example, but the principle remains the same.

2.5 Cause

Definition : Relation where segment B (traditionally called the antecedent) provokes the realization of an event A (the consequent).

Structures :

`Cause` $\rightarrow$
`conn(cause), gap(G), ponct(comma). /`
`conn(cause), gap(G), eos.`

Resources :

conn(cause) : *because, because of, on account of* ponct(comma) : *, ; :*

Examples :

Because books are so thorough and long, you have to learn to skim.
Long lists result in shallow essays because you don't have space to fully explore an idea.
Many poorly crafted essays have been produced on account of a lack of preparation and confidence.

2.6 Condition

Definition : Relation where the segment B refers to a situation which is necessary for A to be realized.

Structures :

Condition →
conn(cond), gap(G), ponct(comma). /
conn(cond), gap(G), eos.

Resources :

conn(cond) : *if*

Examples :

If all of the sources seem to be written by the same person or group of people, you must again seriously consider the validity of the topic.
If you put too many different themes into one body paragraph, then the essay becomes confusing.
For essay conclusions, don't be afraid to be short and sweet if you feel that the argument's been well-made.

2.7 Concession

Definition : Relation where the segment B contradicts part of the segment A, or contradicts the implicit conclusion which can be drawn from segment A.

Structures :

Concession →
conn(opposition_alth), gap(G1), ponct(comma), gap(G2), eos. /
conn(opposition_alth), gap(G), eos. /
conn(opposition_how), gap(G), eos.

Resources :

conn(opposition_alth) : *although, though, even though, even if, notwithstanding, despite, in spite of*
conn(opposition_how) : *however*

Examples :

An essay can be immaculately written, organized, and researched ; however, without a conclusion, the reader is left dumbfounded, frustrated, confused.
Though the word essay has come to be understood as a type of writing in Modern English, its origins provide us with some useful insights.
Your paper should expose some new idea or insight about the topic, not just be a collage of other scholars' thoughts and research – although you will definitely rely upon these scholars as you move toward your point.

2.8 Contrast

Definition : Relation where one segment is opposed to another segment.

Contrast →
conn(opposition_whe), gap(G), ponct(comma). /
conn(opposition_whe), gap(G), eos. /
conn(opposition_how), gap(G), eos.

Resources :

conn(opposition_whe) : *whereas, but whereas, but while*

Examples :

The periodic sentence is one in which the main clause is considerably delayed, whereas the cumulative sentence opens quickly with the main clause.

2.9 Circumstance

Definition : Relation where the segment B refers to a frame in which A is to be realized by the reader of the procedure.

`Circumstance →`
`conn(circ), gap(G), ponct(comma). /`
`conn(circ), gap(G), eos.`

Resources :

conn(circ) : *when, once, as soon as, after, before*

Examples :

Before you put your outline together, you need to identify your argument and analyze it.
Once you use a piece of evidence, be sure and write at least one or two sentences explaining why you use it.

2.10 Purpose

Definition : Relation where segment B provides the aim targeted by the realization of the action expressed in segment A.

`Purpose →`
`conn(purpose), verb(action, infinitive), gap(G), ponct(comma). /`
`conn(purpose), verb(action, infinitive), gap(G), eos.`

Resources :

conn(purpose) : *to, in order to, so as to*

Examples :

To write a good essay on English literature, you need to do five things [...].
In order to make the best of a writing assignment, there are a few rules that can always be followed [...].

2.11 Illustration

Definition : Relation where segment B instantiates a member of segment A, used a representative sample for the class represented by segment A.

`Illustration →`
`exp(illus_eg), gap(G), eos. /`
`[here], auxiliary(be), gap(G1), exp(illus_exa), gap(G2), eos. /`
`[let,us,take], gap(G), exp(illus_bwe), eos.`

Resources :

exp(illus_eg) : *e.g., including, such as*
exp(illus_exa) : *example, an example, examples*
exp(illus_bwe) : *by way of example, by way of illustration*

Examples :

This is a crucial point for other types of writing such as fiction or personal essay writing. Here are some examples of how they can be used well, so long as they are relevant to the essay : [...].

2.12 Illustration : a simple example of a reasoning schema for binding purposes

It is often difficult to identify exactly the text span which is illustrated. As introduced in Chapter 1, in an expression such as *red fruit tart (strawberries, raspberries, etc.)...*, the illustration *(strawberries, raspberries, etc.)* must be properly bound to *red fruit* only. Besides the fact that the two structures (illustration and illustrated) are not adjacent, this relation holds only if it is know that the fruits mentioned are red fruits. If not the relation is rather an elaboration which adds substantial knowledge, contrary to an illustration.

Very informally, the binding rule that binds and illustration with the illustrated texts pan can be defined as follows, assuming here that these are all NPs, with well-identified types :

```
   Illustrate →
<illustrated>, NP(Type), </illustrated>, gap(G),
<illustration> NP1(Type1), NP2(Type2), </illustration>,
[subsume(Type,Type1), subsume(Type, Type2)]
```

The subsumtion control makes sure that the Type of the illustrated is more general than the type of the elements in the illustration.

2.13 Restatement

Definition : Relation where segment B rephrases segment A without adding further information.

```
   Restatement →
ponct(opening_parenthesis), exp(restate), gap(G),
ponct(closing_parenthesis). /
exp(restate), gap(G), eos.
```

Resources :

exp(restate) : *in other words, to put it another way, that is to say, i.e., put differently*

Examples :

If you must say something in a complicated way spanning several sentences, try adding a sentence to summarize the idea. In other words, make every effort possible to be clear about each point in the essay. When you revise your essay, you'll need to ask yourself, is this argument well made ; are there are any gaps in my argument ; am I making the case as precisely as I can ; are there are any premises or points that I make which aren't integrated into the whole paper. In other words, you'll continue to analyze your essay from the organizational and precision perspectives we've already discussed.

2.14 The Art of writing Dislog rules and constraints

The ease of writing rules and the 'natural' character of those rules with respect to language and corpus observations are major properties that any rule system must offer. This however needs experiments over a large number of domains and applications on the way to identify rules, generalize them, reach a certain linguistic adequacy and predictability and elaborate a comprehensive set of linguistic marks, etc. Authoring tools would be useful for various kinds of operations including checking duplicates and overlaps among large sets of rules. While some tools are available for sentence processing (e.g. (Sierra et al. 2008)), there is no such tool customized for discourse. We develop in this section some considerations about a methodology for writing rules and what the services an authoring tool should offer (such a tool is under investigation).

Some investigations have been realized to identify linguistic marks on subsets of discourse relations (Rosner et al 1992), (Redeker, 1990) (Marcu 1997), (Takechi et al. 2003) and (Stede 2012). These mostly establish general principles and methods to extract terms characterizing these relations, rules are then also written by hand (i.e. rules do not result from automatic learning procedures). The linguistic and pragmatic forms and principles that have emerged seem to be compatible with our perspective.

At the present stage, rules are basically written by hand. Although this is not the main trend nowadays, we feel this is the most reliable approach given the complexity and variability of discourse structures and the need to elaborate semantic representations. Let us briefly review here how rules are produced.

The first step is, given a discourse function one wants to investigate, to produce a clear definition of what it exactly represents and what is its scope, possibly in contrast with other functions. This is realized via a first corpus construction where a number of realizations over several domains are collected, analyzed and sorted by decreasing prototypicality order. This must be realized preferably by a few people and in connection with the literature, in order to reach the best consensus.

Then a larger corpus must be elaborated possibly via bootstrapping tools. Morphosyntactic tagging contributes to identifying regularities and frequencies.

From this corpus, a categorization must be first elaborated of the different lexical resources which are needed. Then rules can be written. Rules should be expressed at the right level of abstraction to account for a certain level of predictability and linguistic adequacy. This means avoiding low level rules (one rule per exceptional case) or too high level rules which would be difficult to be constrained. Rules must be well-delimited, starting and ending by non-terminal or terminal symbols which are as specific as possible of the construct. Each rule should implement a particular form of a discourse function. In general, the number of rules for a discourse function (which form a cluster of rules) ranges from 5 to about 25 rules. About 10 are really generic, while the others relate much more restricted situations. This means that managing such a set of rules and evaluating them for a given function on a test corpus is feasible.

The next step is to order rules in the cluster, starting by the most constrained ones considering the processing strategy implemented in <TextCoop>. In general, the most constrained rules correspond to less frequent constructions than generic ones, which could be viewed as the by-default ones. In this case, this means going through a number of rules with little chances of success, involving useless computations. As an alternative, it is possible to start by generic

rules if (1) they correspond to frequently encountered structures and (2) they start by specific symbols not present in the beginning of other rules. In this case, backtracking would occur immediately. This is a compromise, frequently encountered in Logic programming, that needs to be evaluated by the rule author.

Overlap of new rules with already existing ones must be investigated since this will generate ambiguities. This is essentially a syntactic task that requires rule contents inspection. This task could certainly be automated in an authoring tool. Ambiguities may be resolved by using knowledge. If it turns out that this is not possible, then preferences must be stated : a certain function must be preferred to another one. Preferences can then be coded in the cascade, starting with the preferred rule clusters, the recognition of the competing rules being then excluded.

The last stage for rule writing is the development of selective binding rules and possibly correction rules for anomalous situations. Selective binding rules are relatively easy to produce since they are based on the binding of two already identified structures. Structure variability, long-distance or dislocations are automatically managed by the <TextCoop> engine, in a transparent way. Finally, the rule writer must add the cluster name at the right place in the cascade and possibly state constraints as given in Chapter 1.

Although there are important variations, the total investigations for encoding from scratch a discourse function of a standard complexity, including corpus collection, readings and testing should take a maximum of about one month full-time. This is a very reasonable amount of time considering e.g. the workload devoted to corpus annotation in the case of a machine learning approach. We feel the quality of manual encoding is also better, in particular rule authors are aware of the potential weaknesses of their descriptions. If a rule or a small set of rules are already available in an informal way, then encoding this small set in Dislog is much faster : checking for needed lexical resources, writing the rules, checking overlaps and testing the system on a toy text should not take more than a day or two for a somewhat trained person. Our current environment contains about 280 rules describing 16 discourse structures associated with argumentation and explanation. These rules are essentially the core rules for these 16 discourse structures : it is clear that they can be used as a kernel for developing variants or more specific rules for these structures or for structures that share some similarities in form. This should greatly facilitates the development of new rules for trained authors as well as for new ones.

Coming back to an authoring tool, it is necessary at a certain stage to have a clear policy to develop the lexical architecture associated with the rule system. Redundancies (e.g. developing marks for each function even if functions share a lot of them) should be eliminated as much as possible via a higher level of lexical description. This would also help update, reusability and extensions.

Chapitre 3

Using Dislog and TextCoop

3.1 A few warnings before starting

Please consider the following points before starting :
- it is preferable to have some basic knowledge of Prolog before starting to use <TextCoop>,
- we ask you that you do not modify the kernel of the system, we are not responsible of any consequences that may arise if you do so,
- a priori, Prolog does not recognize some characters encoded in UTF8, but only in the ISO formats : some conversions (manual or via a programme) are needed if you have UTF8 encoded texts.
- only the kernel of the system is given here with a few rule samples. There is no interface provided, although this would be desirable. You can however design you own, depending on what you want to do and see.

3.2 Installation

<TextCoop> is implemented in Prolog, only the kernel of Prolog is used, so most versions of Prolog using the Edinburgh syntax should work. We recommend to use SWI Prolog, which is free and runs efficiently on several platforms. Note that it runs faster on a Linux environment than in Windows.

Then, the only thing you have to do is to unzip the <TextCoop> archive into a directory of your choice. A priori it is simpler to keep all the files in a single directory. However, the text files you process can be in a subdirectory.

Basically, the archive contains two directories, one for the French version and the other one for the English version (files end by Fr or Eng depending on the language). Each directory contains the following files :
- the engine : `textcoopV4.pl`
- a specific file for user declarations and parameters : `decl.pl`
- a set of functions : `fonctions.pl`
- a set of lexicons of various terms : `lexiqueBase.pl lexSpecialise.pl, lexiqueIllustr.pl`, you can obviously construct several additional lexicons, you must add their compilation at the beginning of the textcoopV4.pl file. The French version also contains a list of

categorized verbs (eeeaccents.pl).
- a file with rules or patterns : `rules.pl`
- a toy file with 'local' grammar samples written in DCG format : `gram.pl`
- a file for the input-output operations : `es.pl` and for reading files in Prolog : `lire.pl`
- a few files to run the system on examples : `demo.txt`, after processing, the system produces two output files : `demo-out.html` (tags, no colors) and `demo-c-out.html` (same thing but with colors and spaces to facilitate reading). However note that we have not developed at this stage any user interface.
- additional files can be added, for example to include knowledge or specific reasoning procedures. Nothing is included at this level in this pre-beta version.

3.3 Starting

There are several ways to read and modify your files and to call Prolog. Emacs and similar editors are particularly well-adapted. We recommend the use of Editplus V3 for those who do not have any preference. Prolog can be launched directly from the editor and the code can be easily re-interpreted when needed.

To start the system, you must launch Prolog from you environment, e.g. from Editplus. You must then 'consult' your file(s). Since the file `textcoopV4.pl` contains consult orders for all the other files, you just need to consult it, via the menu of the Prolog window, or, in the window by typing :

```
['textcoopV4.pl'].
```

care about ERROR messages, but you can ignore warnings.

It is recommended that the texts you want to analyze are in .txt format. Care to have only ISO character encodings, otherwise Prolog will create huge files via a loop. Character encoding may be tuned in some environments. Then to launch TextCoop, type the main call :

```
annotF.
```

then you are required to enter your file name, between quotes, ended by a dot, as usual in prolog :

```
'demo.txt'.
```

You will then see a large number of intermediate files which are created and re-used. Each cycle corresponds to the execution of a cluster of rules in the rule cascade. Results are stored in two files : `demo-out.html` (html tags, no color) and `demo-c-out.html` (same thing but with colors and spaces to facilitate reading).

The file `es.pl` contains a few other input/output calls that you may wish to explore. You can also change the display colors in this file, or add or withdraw the display of some tags. The contents of the tags is produced by discourse analysis rules, described in the 'Representation' argument.

3.4 Writing rules

Rules are stored in the present archive in the `rules.pl` file. The rules can be produced from a manual analysis of linguistic phenomena or be the result of a statistical analysis. A priori, the Dislog language is flexible enough to accept a large variety of forms. Rule definition

methodology is presented in Chapter 2, together with a number of examples to which the reader can refer.

The first thing, as indicated in the previous chapter, is to make a linguistic analysis, generalize at the appropriate level, develop the lexical resources (cues typical of the relation and other resources) and write rules in an external format, as shown in Chapter 2. We show in this chapter how to write rules in Dislog and to manage clusters of rules, the cascade and various types of constraints.

3.4.1 Writing a rule in Dislog

Let us first take an example. The rule that describes a purpose satellite can be written in an external format as follows :
purpose → connector(goal), verb([action,impinf]), gap(G), punctuation.
e.g. :
To write a good essay on English literature, you need to do five things [...].

This rule says that a purpose satellite is composed of a connector of type goal, followed by an action verb in the imperative or infinitive form followed by a gap. The structure ends by a punctuation mark. Labels such as goal, impinf or action are defined by the rule author, they are not imposed by the system. These tags may be encoded in a variety of ways : as lists (as in this example) or as a feature structure.

The general form of a rule coded in Dislog is :

```
forme(LHS, E, S, RuleBody, Constraints, Result).
```

where :

- LHS is the symbol on the left-hand side of the rule. It is the name of the cluster of rules representing the various structures corresponding to a phenomenon (e.g. purpose satellites). It will be used in the cascade to refer to this cluster and in various constraints.
- E and S and respectiveley the input and output strings of the sentence or text to process, similarly to DCGs. The informal meaning is that between E and S there is a purpose clause. E and S are lists of words as in Prolog.
- RuleBody is the right-hand part of the rule, which is described below,
- Constraints is a list that contains a variety of constraints to check or calls to knowledge and reasoning procedures, these are in general written in Prolog and are executed automatically at the end of the rule analysis. An empty list means no constraints and is always evaluated to true.
- Result denotes the result which includes the string of words of the input structure with tags included at the appropriate places. Tags in rules may include variables.

The rule body is encoded as follows. First, each grammar symbol has four arguments (this is a choice which can be modified, but seems optimal and easy to use). Each symbol has the following form :

```
name(String,Feature,E,S).
```

where :

- String denotes the String which is restored in the result. In general it is the string that has been read for that symbol (e.g. E-S), but it can be any other form (e.g. a normalized form, a reordered string, etc.).
- Feature is the argument that contains a list of restrictions, encoded e.g. as a list of values

or as a type feature structure. The format is here free, but the rule writer must manage it.

– E and S are respectively the input and output lists of words, as above, for the analysis of this particular symbol.

Gap symbols have a different format :

`gap(NotSkipped,Stop,E,S,Skipped).`

where :

– NotSkipped is a list of symbols which must not be skipped before the gap stops (developed in the second argument). If it finds such a symbol, then the gap fails. So far, this list is limited to a single symbol for efficiency reasons. We haven't also found so many cases where multiple restrictions are needed. If really needed, these must be coded (an example is provided in the TextCoop engine code where gaps are coded) in the 'gap' clause in the engine.

– Stop is a list : [Symbol, Restrictions] that describes where the gap must stop : it must stop immediately before it finds a symbol Symbol with the restrictions Restrictions. In general this is the explicit symbol that follows, but this is not compulsory.

– E and S are as above,

– Skipped is the difference between E and S, namely the sequence of words that have been skipped.

It must be noted that a gap must appear in a rule between two explicit symbols. While processing a rule, if a gap reaches the end of a sentence (or a predefined ending mark) without having found the symbols that follows, then it fails.

The symbol skip is slightly different from the gap symbol. It allows to skip a maximum mumber of N words given as parameter. It has the same structure as a gap, except that the second argument is an integer (in general small) :

`skip(NotSkipped, Number, E, S, Skipped)`.

The symbol in the rule that immediately follows the skip symbols defines the termination of the skip.

The rule given above then translates as follows in Dislog :

```
forme(purpose-eng,
E, S,
[connecteur(CONN,goal,E,E1),  verb(V,[action,impinf],E1,E2),
 gap([],[ponct,_],E2,E3,Saute1),  ponct(Ponct,_,E3,S)],
 [],
['<purpose>', CONN, V, Saute1, '</purpose>', Ponct ]).}
```

It is given as a Prolog fact so that the TextCoop meta-interpretor can read it. The reader can note the sequences of input-output variables : E1-E2-E3-S as in DCGs. The last argument encodes the resul, for example the way the original sentence is tagged. Tags may be inserted at any place, they may contain variables elaborated in the inference (also called Constraints) field. In fact, any form of representation can be produced in this field.

Symbols in a rule car be marked as optional or can appear several times, including none. This is encoded via the operators `opt` and `plus` applied to the grammar symbols :

```
forme(purpose-eng,
E, S,
[connecteur(CONN,goal,E,E1),  opt(verb(V,[action,impinf],E1,E2)),
 gap([],[ponct,_],E2,E3,Saute1),  ponct(Ponct,_,E3,S)],
```

```
 [],
['<purpose>', CONN, V, Saute1, '</purpose>', Ponc ]).
```

In this example, the verb is indicated as optional : if it is not found, then the gap starts after the goal marker. If there is no verb, the variable V in the result is not instantiated and does not produce any result.

```
forme(purpose-eng,
E, S,
[connecteur(CONN,goal,E,E1),  plus(aux(Aux,_,E1,E2),
 verb(V,[action,_],E2,E3),
 gap([],[ponct,_],E3,E4,Saute1),  ponct(Ponct,_,E4,S)],
 [],
['<purpose>', CONN, Aux, V, Saute1, '</purpose>', Ponc ]).
```

In this example, a sequence of auxiliaries is allowed before the verb.

3.4.2 Writing context-dependent rules

A closer look at the Dislog rule formalism shows that it is possible to use this formalism to implement context-dependent rules. In fact, the left-hand side symbol, the cluster name, can be just viewed as an identifier, and the power of the rule can be shifted to the pair right-hand part and representation. The right-hand part is indeed often the input form to identify and the Representation is the result, allowing a large diversity of treatments.

Wr already advocated the case of binding rules, which are clearly type 1 if not type 0 rules. It is possble to develop any other kind of rules, e.g. to realize structure transformation with some context sensitivity. If we consider again the example given in Chapter 2 :

```
Warning →
<warning-concl>,gap(G1),</warning-concl>, gap(G2),
<warning-supp>, gap(G3), </warning-supp>, gap(G4), eos.
```

which binds a warning conclusion with a warning support, then the result is a warning, represented by the following XML structure :

```
<warning>
<warning-concl>,G1,</warning-concl>, G2,
<warning-supp>, G3, </warning-supp>, G4,
</warning>.
```

3.4.3 Parameters and Structure Declarations

In contrast with Prolog, but with the aim of improving efficiency, it is necessary in Dislog to declare a few elements. This is realized in the `decl.pl` file. A number of standard symbols are already declared, but check that yours are indeed declared, otherwise your rules will fail.

First in order to allow for a proper variable binding, any symbol used in rules must be declared as follows :

```
tt(adv(Mot,Restr,E,S), E,S).
tt(adj(Mot,Restr,E,S), E,S).
tt(neg(Mot,Restr,E,S), E,S).
```

```
tt(np(Mot,Restr,E,S), E,S).
tt(det(Mot,Restr,E,S), E,S).
```

In this example, the symbols adv, adj, neg, np and det are declared. The co-occurence of the symbols E and S allows to bind the variables of the symbols with the string of words to process in the meta-interpretor (the TextCoop engine).

Similarly, any symbol which can be optional must be declared by means of a piece of code which must be reproduced from the following schema, which encodes the optionality for auxiliaries :

```
opt(aux(AUX,A,E1,E2)) :- aux(AUX,A,E1,E2), !.
opt(aux([],_,E,E)).
```

The same thing is necessary for multiple occurences :

```
plus(adv(T,_,E1,S)) :-
   adv(T1,_,E1,E2), !,
   plus(adv(T2,_,E2,S)), conc(T,S,E1).
plus(adv(T,_,S,S)).
```

This portion of code must be duplicated for all relevant symbols. A higher-order encoding could have been realized, but it does affect efficiency quite substantially.

Constraints presented in chapter 1 must also be declared (at least one instance to avoid failures), a few examples are provided here :

```
exclut_unite(title).

termin(['<',condition,'>'],['<','/',condition,'>']).
termin(['<',purpose,'>'],['<','/',purpose,'>']).
termin(['<',circumstance,'>'],['<','/',circumstance,'>']).
termin(['<',restatement,'>'],['<','/',restatement,'>']).

dom(instr-eng,[but, condopt-eng, restatement]).

non_dom(instr-eng,[avt,cons]).
```

A few constraints are given in the decl.pl file as examples. These must be kept to avoid system failures.

This file also contains the cascade declaration, as explained below.

3.4.4 Lexical data

Lexical data is specified in the `lexique.pl` file. Lexical data can follow the stardard categories and features of linguistic theories or be ad'hoc, depending on specific situations. Lexical data is given in DCG format. You have to design yourself you own lexicon. In this first version we simply provide a few examples. However, we are developing sets of lexical markers and other resources which are useful for discourse analysis. These will be made available in a second version.

Here are a few examples included into this version :

```
 % pronouns
pro([we],_) --> [we].
pro([you],_) --> [you].

 % goal connectors
connecteur([in,order,to], goal) --> [in,order,to].
connecteur([in,order,that], goal) --> [in,order,that].
connecteur([so], goal) --> [so].
connecteur([so,as,to], goal) --> [so,as,to].

 % a few specific marks describing the beginning of a sentence
mdeb([debph],_) --> [debph]. % internal mark
mdeb(['<li>'],_) --> ['<',li,'>'].
mdeb([1],_) --> ['-',1].

 % condition
expr([if],cond) --> [if].

 % specific marks for reformulation
expr_reform([in,other,words],_) --> [in,other,words].
expr_reform([to,put,it,another,way],_) --> [to,put,it,another,way].
expr_reform([that,is,to,say],_) --> [that,is,to,say].

 % tag lexical data: this is useful for rules
 % which basically bind structures on the basis of already produced tags,
 % each elements has a type specified in the second argument.
balise([<, instruction, >],instruction) --> [<, instruction, >].
balise([<, /, instruction, >],endinstruction) --> [<, /, instruction, >].

balise([<, goal, >],goal) --> [<, goal, >].
balise([<,/,goal, >],endgoal) --> [<, /,goal, >].
```

3.4.5 Other resources

This first version is relatively limited and does not contain so many additional tools and facilities. These will come in the second version of the tool. However, it contains the kernel necessary to implement the recognition of most discourse structures and to bind them.

The file `gram.pl` contains a few grammar rules written in DCG style and compatible with the symbol format given above. Indeed, it is possible to have symbols in rules which are non-terminal and which are associated with a grammar in that module :

```
np([A,B],_,E,S) :- det(A,_,E,S1), n(B,_,S1,S).

np([A],_,E,S) :- pro(A,_,E,S).
```

This short sample of a grammar for nps can be used in Dislog rules as such. Note that the first argument of the np symbols contains the string of words which have been processed. This argument could include any other form, e.g. a normalized form or a tree.

3.4.6 Input-Output management

The input-output file management is realized in several files. The main file is `es.pl` which contains the main calls, dynamically produces names for output and intermediate files (to avoid conflicts between parses) and produces two kinds of displays : a file for further processing which is a basic XML file and a similar file where structures get a color for easier reading. This latter file can be read by various XML editors. File names are created dynamically : `demo-out.html` (tags, no colors) and `demo-c-out.html` (tags and colors).

If you have sufficient Prolog programming skills, you can modify this file, e.g. changing colors, as you need it.

It is important to note that, in this first version, structure processing is realized on a sentence basis. We have improved and parameterized this situation which is somewhat limited, an extension is available on demand. Meanwhile, you can end the text portions you want to process by a dot, and replace dots ending sentences in these portions by another symbol, e.g. the word 'dot', which can be re-written later by a real dot in the output file.

Basically, input files must be plain text, possibly with XML marks. Word files cannot be processed. It must also be noted that Prolog has some difficulties with UTF8 encoded texts.

The two other files for input-output operations are internal to the system and should not be modified : `lire.pl` reads files under various formats and produces a list of words, which is the entry for the main processing. In this module, some characters are transformed into different codes in order to avoid any interference with Prolog predefined elements. These are then restored in their original form when the final output form is produced. This is an important issue to keep in mind, since some elements in teh elxicon must take these aspects into account.

The module `functions.pl` contains a variety of basic utilities, which you may use for various purposes besides the present software.

3.5 Execution schema and structure of control

The <TextCoop> engine is a meta-interpretor written in standard Prolog. This is a well-know technique in Logic Programming which is very convenient for developing e.g. alternative processing strategies or demonstrators. The strategy implemented in <TextCoop> is quite similar to the Prolog strategy. However there are some major differences you need to be aware of.

The <TextCoop> engine will consider, for a given text, rule clusters one after the other. Therefore, rule clusters must all be organized in a cascade that describes the cluster execution order. It must be declared by a cascade (as an automata cascade) in the `decl.pl` file as follows :

```
cascade(eng, [circ-eng, condition-eng,
  purpose-eng, restate-eng, illus-eng]).
```

The whole text is inspected for each rule cluster, one after the other. If a cluster does not produces any result, then, the next one is activated, there is no failure. In case you wish to define several cascades, the first argument of the cascade predicate is its identifier ('eng' here). The maximal length of a cascade is 60 elements, which sounds quite large. During execution, you can see in the Prolog window the different steps with the intermediate files being compiled.

Within a cluster, rules are considered one after the other, from the first to the last one, similarly to Prolog strategy. However, there is here a major difference. The string of words to process is traversed from left to right (this code also provides a right to left strategy), at each step (i.e. for each word), the engine attempts to find a rule in that cluster that would start by this word (via derivation or lexical inspection). If it works, then that rule, independently of its position in the cluster is activated. If the whole rule succeeds, then no other rule is considered in that cluster, otherwise backtracking occurs.

For example, consider the following sentence to process : [a,n,a,d,f,b,c], and the following (simplified) set of rules :

```
s --> d, f.
s --> a, b.
s --> a, d.
```

The first a and then the n are considered without any success, but then the second occurence of a is the left-corner of the second and third rules. The second fails since no b is found after a, but the third rule succeeds. Note that in DCGs the first rule would have succeeded on a partial parse because the sentence contains the sequence [d,f], but since it comes later than the sequence [a,d], it does not succeed in Dislog. This strategy favors left-extraposed structures in case there are several of them in a sentence. Note also that if the second rule where : s $\rightarrow$ a, gap, b. then it would have succeeded first on the segment [a,n,a,d,f,b] with gap = [n,a,d,f].

In our approach, when a rule in a cluster succeeds, for efficiency reasons, no other rule in that cluster is considered any more and there is no backtraking at any further stage. For users who want to recognize several occurences of the same structure in a sentence, it is best to have a more complex rule that repeats the structure to find. This is not so elegant, but allows to limit backtracking and entails a much better efficiency. It is also possible to suppress a ! in the code, but this is not recommended.

Bibliographie

[1] Bal, B.K., Saint-Dizier, P., 2010. Towards Building Annotated Resources for Analyzing Opinions and Argumentation in News Editorial, LREC, Malta.

[2] Carberry, S., 1990. Plan Recognition in natural language dialogue, Cambridge university Press, MIT Press.

[3] Carlson, L., Marcu, D., Okurowski, M.E., 2001. Building a Discourse- Tagged Corpus in the Framework of Rhetorical Structure Theory. In Proceedings of the 2nd SIGdial Workshop on Discourse and Dialog, Aalborg.

[4] Colmerauer, A., 1978. Metamorphosis Grammars, in Natural language understanding by computers, L. Bolc (ed.), LNCS no. 63, Springer verlag.

[5] Delin, J., Hartley, A., Paris, C., Scott, D., Vander Linden, K., 1994. *Expressing Procedural Relationships in Multilingual Instructions*, Proceedings of the Seventh International Workshop on Natural Language Generation, pp. 61-70, Maine, USA.

[6] Di Eugenio, B. and Webber, B.L., 1996. Pragmatic Overloading in Natural Language Instructions, *International Journal of Expert Systems*.

[7] Fontan, L., Saint-Dizier, P., 2008. Analyzing the explanation structure of procedural texts : dealing with Advices and Warnings, International Symposium on Text Semantics (STEP 2008), Venise, , Johan Bos (Eds.).

[8] Grosz, B., Sidner, C., 1986. Attention, intention and the structure of discourse, Computational Linguistics 12(3).

[9] Kintsch, W., 1988. *The Role of Knowledge in Discourse Comprehension : A Construction-Integration Model*, Psychological Review, vol 95-2.

[10] Kosseim, L., Lapalme, G., 2000. *Choosing Rhetorical Structures to Plan Instructional Texts*, Computational Intelligence, Blackwell, Boston.

[11] Lasnik, H., Uriagereka, J., 1988. *A Course in GB syntax*, MIT Press.

[12] Mann, W., Thompson, S., 1988. Rhetorical Structure Theory : Towards a Functional Theory of Text Organisation, *TEXT* 8 (3) pp. 243-281.

[13] Mann, W., Thompson, S.A. (eds), 1992. Discourse Description : diverse linguistic analyses of a fund raising text, John Benjamins.

[14] Marcu, D., 1997. The Rhetorical Parsing of Natural Language Texts, ACL'97.

[15] Marcu, D., 2000. *The Theory and Practice of Discourse Parsing and Summarization*, MIT Press.

[16] Marcu, D., 2002. An unsupervised approach to recognizing Discourse relations, ACL'02.

[17] Miltasaki, E., Prasad, R., Joshi, A., Webber, B., 2004. Annotating Discourse Connectives and Their Arguments, proceedings of new frontiers in NLP.

[18] Pereira, F., 1981. Extraposition Grammars, *Computational Linguistics*, vol. 9-4.

[19] Pereira, F., Warren, D., 1980. Definite Clause Grammars for Language Analysis, *Artificial Intelligence* vol. 13-3.

[20] Rosner, D., Stede, M., 1992. *Customizing RST for the Automatic Production of Technical Manuals*, in R. Dale, E. Hovy, D. Rosner and O. Stock eds., Aspects of Automated Natural Language Generation, Lecture Notes in Artificial Intelligence, pp. 199-214, Springler-Verlag.

[21] Saaba A., Sawamura, H., 2008. Argument Mining Using Highly Structured Argument Repertoire, *proceedings EDM08*, Niigata.

[22] Saito, M., Yamamoto, K., Sekine, S., 2006. Using Phrasal Patterns to Identify Discourse Relations, ACL'06.

[23] Saint-Dizier, P., 1994. *Advanced Logic programming for language processing*, Academic Press.

[24] Saint-Dizier, P., 2012. Processing Natural Language Arguments with the <TextCoop> Platform, *Journal of Argumentation and Computation*, vol 3-1.

[25] Takechi, M., Tokunaga, T., Matsumoto, Y., Tanaka, H., 2003. *Feature Selection in Categorizing Procedural Expressions*, The Sixth International Workshop on Information Retrieval with Asian Languages (IRAL2003), pp.49-56.

[26] Van Dijk, T.A., 1980. *Macrostructures*, Hillsdale, NJ : Lawrence Erlbaum Associates.

[27] Webber, B., 2004. D-LTAG : extending lexicalized TAGs to Discourse, Cognitive Science 28, pp. 751-779, Elsevier.

[28] Wierzbicka, A., 1987. English Speech Act Verbs, Academic Press.